Modular Verification of Termination and Execution Time Bounds Using Separation Logic

Jafar Hamin *Bart Jacobs*

Report CW 696, April 2016

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Modular Verification of Termination and Execution Time Bounds Using Separation Logic

Jafar Hamin *Bart Jacobs*

Report CW 696, April 2016

Department of Computer Science, KU Leuven

Abstract

This document presents a formal method to verify execution time bounds of programs at the source level, where timing constraints along with other functional requirements are specified in the routines' contracts and are verified in a modular manner. The approach works based on a countdown time budget mechanism to guarantee the termination of the input program, and incorporates the concepts of separation logic, making it more integrable with verification approaches for pointer-manipulating programs and applicable for concurrent programs where time resource needs to be passed among different threads. We selected the MSP430 microcontroller as well as a simple non-optimizing compiler as a case study and defined a co-inductive concrete semantic to model time consumption and potential non-termination of commands based on this platform. Accordingly, we developed the corresponding symbolic execution and proved that it is sound, i.e., if a program does not fail in the symbolic execution, then it respects the specified time bounds in the concrete execution too. Our preliminary results show that the proposed approach can be used to verify time bounds of programs involving separation logic based specifications.

Modular Verification of Termination and Execution Time Bounds Using Separation Logic

Jafar Hamin
iMinds-DistriNet, Dept. C.S., KU Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
jafar.hamin@cs.kuleuven.be

Bart Jacobs
iMinds-DistriNet, Dept. C.S., KU Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
bart.jacobs@cs.kuleuven.be

Abstract

This paper presents a formal method to verify execution time bounds of programs at the source level, where timing constraints along with other functional requirements are specified in the routines' contracts and are verified in a modular manner. The approach works based on a countdown time budget mechanism to guarantee the termination of the input program, and incorporates the concepts of separation logic, making it more integrable with verification approaches for pointer-manipulating programs and applicable for concurrent programs where time resource needs to be passed among different threads. We selected the MSP430 microcontroller as well as a simple non-optimizing compiler as a case study and defined a co-inductive concrete semantic to model time consumption and potential non-termination of commands based on this platform. Accordingly, we developed the corresponding symbolic execution and proved that it is sound, i.e., if a program does not fail in the symbolic execution, then it respects the specified time bounds in the concrete execution too. Our preliminary results show that the proposed approach can be used to verify time bounds of programs involving separation logic based specifications.

1 Introduction

Hard real-time requirements of the tasks in safety-critical embedded systems have resulted in numerous research investigating techniques to estimate timing specifications of such systems in terms of worst-case execution time (WCET). These studies mostly fall into two main categories; measurement based ones, that attempt to approximate WCET of a program by executing it under different inputs and states, and static analysis of execution time [5, 10] in which this approximation is based on an abstract model of the machine and is obtained as the result of three analysis phases: 1) control flow analysis to identify flow of execution [3, 9], 2) process behavior analysis that deals with hardware components which influence the program timing, and 3) estimate calculation that computes the global bound based on the results of the previous phases.

The result of such analysis, however, needs to be formally verified, particularly, when it comes to critical applications where a reliable guarantee is demanded. Several attempts have been made to provide a (time) bound with such guarantee [1, 4, 6]. Elvira et al. (2011), for example, gives this guarantee by combining COSTA, a static analyzer for inferring upper (time) bounds of routines and loops, with KeY, a source code verification tool for Java programs, such that the result of the former is verified by the latter. The state-of-the-art verification tools, however, in addition to routines' contracts and loop invariants, mostly require extra annotations, namely *variant* (or *decrease*) clauses, and extra checks to verify validity of such propositions. Additionally, there is no clear way to verify code involving multi threading where *time budget* needs to be passed among these modules.

In this paper we incorporate the concepts used in separation logic [12] such as heap, production, consumption and exchange of heap chunks to verify time bounds of programs where timing constraints along with other properties are specified in the contract of routines and loops. In addition to having a unified mechanism to verify both functional and non-functional specifications, it makes our approach highly integrable with verification approaches for pointer-manipulating programs allowing it to benefit from other propositions that describe the shape of data structures in the memory. More specifically, we present a specification formalism as well as a verification algorithm such that given a high-level program as the input, if the verification algorithm reports that each module of the program satisfies its contract, then the whole program satisfies the specified time bound. We instantiate this formalism according to the microcontroller *MSP430* [7] and our non-optimizing compiler *Prext* (Predictable execution time) to define a co-inductive

concrete semantics modeling time consumption and potential non-termination of commands and then prove the soundness of our algorithm. The formal development presented in this paper is based on the formalization and soundness proof of Featherweight VeriFast [14], where functional properties of imperative programs are modularly verified against user-provided annotations.

The remainder of the paper is organized as follows: Section 2 provides an overall overview of the approach. Section 3 provides a definition of concrete execution of the high-level source code. The corresponding symbolic execution is then introduced in section 4. We provide a soundness proof of our approach in section 5. In section 6 we report on experiment and then in section 7 we draw conclusions.

2 An Informal Overview

This section provides an informal overview of how the presented work verifies the time bound specified in the contract of the routines and loops. Consider routine $\mathsf{foo}(\mathbf{n})$ claiming to take at most $19{\times}\mathbf{n}{+}70$ cycles to execute where $\mathbf{n}$ is required to be non-negative. The symbolic execution of this routine, as indicated in the following, starts by initializing the *state* of the execution (step 1). The state is composed of three components; namely *store* (s), *heap* (h) and *path condition* (Φ). The store maps each variable of the program to a *term*. A term can be a literal number, a *symbol* that represents an arbitrary value, as well as addition, subtraction or multiplication of terms. The path condition is a set of *formulae* representing equalities and inequalities between terms. Heap is a multi-set of *chunks* that maps each potential element to a term representing the number of times it occurs in the multi-set. We introduce chunk tb representing availability of one unit of time (cycle), and hence, $h(\mathsf{tb})$ indicates the number of cycles that can be spent to execute the rest of the commands in the routine. In the initialized state the heap is empty ($\mathbf{0}$) and the store binds the input parameter(s) of the routine to *fresh* symbol(s) and every other variable to zero ($\mathbf{0}[\mathbf{n}{:}n]$). After initializing the state, the next step (2) is to *produce* the *precondition* of the routine denoted by $\mathbf{req}\ a$, where a is an *assertion* and can be a boolean expression, availability of e amount of time budget denoted by $[e]\mathsf{tb}$, where e is a (mathematical) expression, as well as a separating conjunction of assertions denoted by $*$. For producing a boolean expression, the expression first gets evaluated to a formula by replacing all its free variables by their corresponding terms in the store, and then either is added to the path condition if it is consistent with the formulae in the path condition or leads the symbolic execution to successfully finish the verification of the routine if it contradicts the path condition's formulae. When producing e amount of time budget (tb), e is first evaluated to term t, and if the formula $0{\leqslant}t$ does not contradict the path condition, t is added to the time budget by increasing $h(\mathsf{tb})$ by t. Otherwise, the verification of the routine successfully finishes. Production of a conjunction assertion involves producing all assertions in the conjunction.

$\mathbf{routine}\ \mathsf{foo}(\mathbf{n})$
  $\mathbf{req}\ [19{\times}\mathbf{n}{+}70]\mathsf{tb} * (0{-}1){<}\mathbf{n}\quad \mathbf{ens}\ 1{=}1$
    //1) intitialize the state
    $s{:}\mathbf{0}[\mathbf{n}{:}n], h{:}\mathbf{0}, \Phi{:}\{\}$
    //2) produce precondition
    $s{:}\mathbf{0}[\mathbf{n}{:}n], h{:}\{\!|(19{\times}n{+}70){\cdot}\mathsf{tb}|\!\}, \Phi{:}\{-1{<}n\}$
    //3) consume $[5]\mathsf{tb}$ for entering the routine
    $s{:}\mathbf{0}[\mathbf{n}{:}n], h{:}\{\!|(19{\times}n{+}65){\cdot}\mathsf{tb}|\!\}, \Phi{:}\{-1{<}n\}$
{ $\mathbf{i} := 0$;
    //4) update the store and consume $[5]\mathsf{tb}$ for assignment
    $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}0], h{:}\{\!|(19{\times}n{+}60){\cdot}\mathsf{tb}|\!\}, \Phi{:}\{-1{<}n\}$
    //5) consume the invariant
    $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}0], h{:}\{\!|51{\cdot}\mathsf{tb}|\!\}, \Phi{:}\{-1{<}n\}$
    //6) havoc targets of the loop body
    $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}i], h{:}\{\!|51{\cdot}\mathsf{tb}|\!\}, \Phi{:}\{-1{<}n\}$
  $\mathbf{while}\ \mathbf{i}{<}\mathbf{n}\ \mathbf{inv}\ \mathbf{i}{<}\mathbf{n}{+}1 * 0{-}1{<}\mathbf{i} * [19{\times}(\mathbf{n}{-}\mathbf{i}){+}9]\mathsf{tb}$
      //7-1) empty the heap
      $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}i], h{:}\mathbf{0}, \Phi{:}\{-1{<}n\}$
      //7-2) produce the invariant and the guard
      $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}i], h{:}\{\!|(19{\times}(n{-}i){+}9){\cdot}\mathsf{tb}|\!\},$
      $\Phi{:}\{-1{<}n, i{<}n{+}1, -1{<}i, i{<}n\}$
      //7-3) consume $[9]\mathsf{tb}$ for checking the guard
      $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}i], h{:}\{\!|(19{\times}(n{-}i)){\cdot}\mathsf{tb}|\!\}, \Phi{:}\{unch\}$
   $\mathbf{do}\{\ \mathbf{i} := \mathbf{i}{+}1$
      //7-4) consume $[10]\mathsf{tb}$ for assignment and jump
      $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}i{+}1], h{:}\{\!|(19{\times}(n{-}i){-}10){\cdot}\mathsf{tb}|\!\}, \Phi{:}\{unch\}$
      //7-5) consume invariant
      $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}i{+}1], h{:}\{\!|0{\cdot}\mathsf{tb}|\!\}, \Phi{:}\{unch\}$
      //7-6) execution path ends
  }; //8-1) continue from 6 and produce the invariant
      $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}i], h{:}\{\!|(51{+}19{\times}(n{-}i){+}9){\cdot}\mathsf{tb}|\!\},$
      $\Phi{:}\{-1{<}n, i{<}n{+}1, -1{<}i\}$
      //8-2) produce $\neg$ guard and consume $[9]\mathsf{tb}$
      $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}i], h{:}\{\!|(51{+}19{\times}0){\cdot}\mathsf{tb}|\!\},$
      $\Phi{:}\{-1{<}n, i{<}n{+}1, -1{<}i, \neg(i{<}n)\}$
  $\mathsf{bar}(1)$
    //9) consume $[7]\mathsf{tb}$ to evaluate arg and call the routine
    $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}n{+}4], h{:}\{\!|44{\cdot}\mathsf{tb}|\!\}, \Phi{:}\{unchanged\}$
    //10) consume the precondition of the routine bar
    $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}n{+}4], h{:}\{\!|24{\cdot}\mathsf{tb}|\!\}, \Phi{:}\{unchanged\}$
    //11) produce the postcondition of the routine bar
    $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}n{+}4], h{:}\{\!|24{\cdot}\mathsf{tb}|\!\}, \Phi{:}\{unchanged\}$
}   //12) consume $[7]\mathsf{tb}$ for leaving the routine
    $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}n{+}4], h{:}\{\!|17{\cdot}\mathsf{tb}|\!\}, \Phi{:}\{unchanged\}$
    //13) consume the postcondition
    $s{:}\mathbf{0}[\mathbf{n}{:}n, \mathbf{i}{:}n{+}4], h{:}\{\!|17{\cdot}\mathsf{tb}|\!\}, \Phi{:}\{unchanged\}$

After producing the precondition, the next step (3) is to consume some amount of time budget for the instructions

that should be executed at the beginning of the routine; adjusting the stack pointer, for example. When consuming e amount of tb, e first gets evaluated to t, and if the path condition (more precisely, the conjunction of formulae in the path condition) implies that $0 \leqslant t$ and $t \leqslant h(\mathsf{tb})$ then this amount is subtracted from the current budget. Otherwise verification of the routine fails. Symbolic execution of an assignment statement (4), in addition to consuming the required number of cycles, evaluates the right hand side expression to a term and then updates the store by binding the left hand side variable to this term.

Symbolic execution of a loop statement involves two separate phases. The first phase only aims to verify the invariant, i.e., that it holds before and after each iteration of the body. Then in the second phase, instead of executing the loop, the invariant is exploited by consuming and producing it to apply the effect of the execution of the loop on the state. Note that for consuming a boolean expression, the expression is first evaluated to a formula and if the result can be proven from the path condition, the execution just continues. Otherwise the verification of the routine fails. To verify the invariant in the first phase, all of the variables whose values may change during the execution of the body (target variables) are first detected and then havocked by updating their values in the store and binding them to fresh symbols (6). The heap is emptied (7-1) and both invariant and loop guard are produced (7-2). Then the time needed for evaluating the loop guard is consumed and the rest of the commands get executed until reaching the end of the body. At this point, the invariant is consumed to make sure that it holds at the end of each iteration (7-5). If it does not fail the invariant has been verified and the execution of this phase ends (7-6) and the second phase will start. Notice that in the example the time budget at this step is zero, meaning that the invariant has an exact estimation of the time required for execution of the loop. In the second phase, the invariant is consumed (5) and again all target variables of the loop are havocked. It then produces the invariant, this time with new fresh symbols bound to the target variables (8-1). As a consequence, the rest of the execution does not rely on the old values of the target variables. After consuming the time needed for checking the guard, the negation of the guard is produced and the second phase ends (8-2). The rest of the execution continues from the state resulting from the execution of the second phase. Notice that at this step, Φ implies that n and i are equal, hence, the current time budget does not depend on $\mathbf{n}$ anymore.

Symbolic execution of a routine call first consumes time needed for evaluating the arguments and preparing the registers to call the routine (9) and then instead of executing the callee's body, it simply consumes its precondition (10) and then produces its postcondition (11). In this example we assume that the routine $\mathsf{bar}(\mathbf{n})$ has a trivial postcondition and $[20\times\mathbf{n}]\mathsf{tb} * 0<\mathbf{n}$ as its precondition, that can be consumed without any failure. As the last step, after consuming some amount of time for leaving the routine (12), the postcondition in the contract is consumed (13). If this consumption does not fail it means that the verification of the routine has been successfully finished. Notice that after this consumption there is still 17 budget left in the heap (leak in time), meaning that the contract has overestimated its time requirement. We allow this overestimation, however, it is even possible to prevent that by checking that the remaining time budget is zero at the end of the execution.

Having this background, in the subsequent sections we provide a formal system for our approach starting by introducing the notion of *concrete execution* that reflects the behavior of programs when executing on the machine.

3 Concrete Execution

To have a formal definition of concrete execution of the commands we start by defining a simple programming language as follows where c is a command:

$$\begin{array}{rl} e_1 ::= & z \mid x \text{ where } x \in \mathit{Vars},\ z \in \mathbb{Z} \\ e ::= & e_1 \mid e_1 + e_1 \mid e_1 - e_1 \\ b ::= & e = e \mid e < e \\ c ::= & x := e \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \\ & \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid r(e) \mid c; c \end{array}$$

Execution. The execution is defined as a function; given a command it returns another function, namely a *mutator*, that maps an initial execution *state* to an *outcome*. This way we ease the definition of execution by composing and reusing the mutators. It also helps with proving the soundness of the symbolic execution, where instead of commands we deal with mutators. In the concrete execution a state (σ) is a *store* that maps each program variable to an integer value.

$$\begin{array}{rl} \mathit{CStores} = & \mathit{Vars} \to \mathbb{Z} \\ \mathit{CStates} = & \mathit{CStores} \\ \mathit{CMutators} = & \mathit{CStates} \to \mathit{COutcomes} \\ \mathsf{exec} \in & \mathit{Commands} \to \mathit{CMutators} \end{array}$$

Outcomes. The set of outcomes is defined co-inductively; an outcome is a singleton outcome, a demonic or angelic choice among a set of outcomes, or a step outcome indexed by an integer duration value. A singleton outcome is a pair of a post-state and an *answer* that is an optional value of a generic type. This enables mutators to pass a value to the subsequent mutator when they are sequentially composed. The notion of demonic choice among outcomes aims to cover the execution of all possible traces of the program, where an execution decision is based on information

that remains unrevealed until run time. A conditional statement, for example, leads to a demonic choice between resulting outcomes of execution of its branches, and it is demonic because all branches are verified to demonically find an unintended timing behavior. Although the notion of angelic choice is introduced in this section, it is applicable in the symbolic execution where an angelic choice among an empty set leads to a verification failure ($\bot$). Analogously, a demonic choice over an empty set results in a verification success ($\top$). The index in the step outcome represents the timing behavior of the command. More specifically, it indicates the number of machine cycles taken by the command to get executed on the machine. Note that execution of a command may result in an outcome with multiple steps. Execution of an assignment statement, for example, leads to an outcome having two steps (that may have different indexes), one for evaluating the right-hand side expression and another for binding the result to the left-hand side variable. We allow a negative index for the sake of the soundness proof where we relate a symbolic execution state to its corresponding concrete one. Since the set of outcomes is defined co-inductively [11], it supports commands with an infinite number of steps.

$$
\begin{array}{rcll}
\phi & ::= & \langle \sigma, a \rangle & \text{singleton outcome} \\
& | & \bigotimes \Phi & \text{indexed demonic choice} \\
& | & \bigoplus \Phi & \text{indexed angelic choice} \\
& | & \Uparrow_z \phi & \text{step outcome}
\end{array}
$$

Sequential Composition. To sequentially compose the mutators, we first define sequential composition of an outcome to a mutator where state and answer of the former is passed to the latter resulting in another outcome. Composing a demonic or angelic choice over a set of outcomes and a mutator is a demonic or angelic choice among sequential composition of each outcome to that mutator, respectively. In the case of an outcome with a step, the inner outcome composes to the mutator and then the step is appended to the result. With the assistance of this definition, sequential composition of mutators is defined as a mutator where given a state, the outcome produced by passing this state to the first mutator is sequentially composed to the second mutator. This concept can also be generalized to cover outcomes with answers, where in the composition of $x \leftarrow \phi; C(x)$ the answer of outcome ϕ, that is bound to x, is the input of $C(x)$ that is a function from answers to mutators. We also define a new kind of sequential composition, denoted by $C;,C'$, where after executing C and then C', its result is the result of C.

$$
\begin{array}{rcl}
\langle \sigma \rangle; C & = & C(\sigma) \\
(\bigotimes \Phi); C & = & \bigotimes \{\phi \in \Phi.\ (\phi; C)\} \\
(\bigoplus \Phi); C & = & \bigoplus \{\phi \in \Phi.\ (\phi; C)\} \\
(\Uparrow_z \phi); C & = & \Uparrow_z (\phi; C) \\
C; C' & = & \lambda\sigma.\ C(\sigma); C'
\end{array}
$$

Generalizing Demonic and Angelic Choice. A demonic choice among a set of mutators $\tilde{C}$ is defined as a mutator, that for a given input state, demonically chooses among the outcomes obtained by passing this state to the mutators in $\tilde{C}$. An angelic or demonic choice over a boolean expressionis defined as follows:

$$
\begin{array}{ll}
\bigotimes \tilde{C} = \lambda\sigma.\ \bigotimes \{C \in \tilde{C}.\ C(\sigma)\} & \\
\bigotimes \mathsf{true}.\ \phi = \phi & \bigotimes \mathsf{false}.\ \phi = \top \\
\bigoplus \mathsf{true}.\ \phi = \phi & \bigoplus \mathsf{false}.\ \phi = \bot
\end{array}
$$

Satisfaction. We inductively define the satisfaction relationship relating an outcome and an desirable postcondition; a set of state-answer pairs indexed by an integer value that indicates the time bound. Since this relationship is inductively defined infinite outcomes never satisfiy the postcondition. A singleton outcome satisfies a condition if it is a member of that condition under any non-negative time bound. A demonic choice among a set of outcomes satisfies a condition if each outcome in the set satisfies that condition. For an angelic choice, satisfaction of at least one of the outcomes in the set would suffice. An outcome ϕ indexed by z step(s) satisfies a postcondition with time bound t, if ϕ satisfies that postcondition under time bound $t-z$.

$$
\begin{array}{rcl}
\langle \sigma, a \rangle\ \{Q\}_{t+0} & \Leftrightarrow & (\sigma, a) \in Q \\
\bigotimes \Phi\ \{Q\}_t & \Leftrightarrow & \forall \phi \in \Phi.\ \phi\ \{Q\}_t \\
\bigoplus \Phi\ \{Q\}_t & \Leftrightarrow & \exists \phi \in \Phi.\ \phi\ \{Q\}_t \\
\Uparrow_z \phi\ \{Q\}_t & \Leftrightarrow & \phi\ \{Q\}_{t-z}
\end{array}
$$

Timing Behavior. The timing behavior of a piece of code definitely depends on the way that it gets compiled and the machine on which the corresponding binaries are executed. In the last few years there has been an interest to bridge this gap [2, 13]. Amadio et al. (2014), for example, introduce a mechanism in which in addition to compiling the source code, the compiler provides some extra information about timing behavior of the compiled code. For this research, however, it is assumed that this information is available and we base our concrete execution on this information. As a case study we use our non-optimizing compiler Prext that follows specific patterns to compile high-level source code into the machine code readable by the MSP430, a microcontroller that has no cache or pipeline mechanism. This way we can introduce the function cycle, that given an expression e specifies the number of cycles taken to evaluate e as follows:

$$
\begin{array}{c}
\mathsf{cycle}(z){=}2 \quad \mathsf{cycle}(x){=}3 \\
\mathsf{cycle}(e_1{+}e_2){=}\mathsf{cycle}(e_1{-}e_2){=}\mathsf{cycle}(e_1){+}\mathsf{cycle}(e_2) \\
\mathsf{cycle}(e_1{=}e_2){=}\mathsf{cycle}(e_1{<}e_2){=}\mathsf{cycle}(e_1){+}\mathsf{cycle}(e_2)
\end{array}
$$

Auxiliary Mutators. According to the timing behavior of the source code it is now possible to define the auxiliary

mutators that are used in the definition of the concrete execution. The mutator assume_0 parametrized by a boolean expression b, evaluates b in the current store s, denoted by $[\![b]\!]_s$, and if the result is false it leads to an unreachable outcome. Otherwise it has no effect on the execution. The mutator assume is similarly defined but loads an extra step for evaluating and comparing the expressions in b. The mutator $\mathsf{eval}(e)$ evaluates e and returns it as the result. It also imposes the required time for this evaluation. The assignment mutator updates the value of the left-hand side variable in the store and loads three extra cycles onto the outcome. The mutators jump, call, enter and leave have no side effect and only load the number of cycles they need for jumping, calling, entering and leaving a routine, respectively. The mutator store simply returns the current store. A store assignment mutator replaces the current store with the new one. The mutator with, given a store s' and a mutator C', without affecting the initial store, temporarily executes C' in the store s'. The mutator C^* is a demonic choice among executions of $C;\mathsf{jump}$ with any arbitrary number of iterations.

$$
\begin{aligned}
\mathsf{assume}_0(b) &= \lambda\ s.\ \bigotimes [\![b]\!]_s = \mathsf{true}.\ \langle s \rangle \\
\mathsf{assume}(b) &= \lambda\ s.\ \bigotimes [\![b]\!]_s = \mathsf{true}.\ \Uparrow_{\mathsf{cycle}(b)+1}\ \langle s \rangle \\
\mathsf{eval}(e) &= \lambda\ s.\ \Uparrow_{\mathsf{cycle}(e)}\ \langle s, [\![e]\!]_s \rangle \\
\mathsf{store} &= \lambda\ s.\ \langle s, s \rangle \\
\mathsf{store} := s' &= \lambda\ s.\ \langle s' \rangle \\
\mathsf{with}(s', C) &= s \leftarrow \mathsf{store}; \mathsf{store} := s'; \mathsf{C};, \mathsf{store} := s \\
x := v &= \lambda\ s.\ \Uparrow_3 \langle s[x := v] \rangle \\
\mathsf{jump} &= \lambda\ s.\ \Uparrow_2 \langle s \rangle \\
C^* &= \mathsf{noop} \otimes C; \mathsf{jump}; C^*
\end{aligned}
$$

Execution of Commands. As previously mentioned, the execution is a function that given a command results in a mutator. Execution of an assignment is a sequential composition of two mutators; the first one evaluates the expression and passes the result to the second one that binds the variable to the new value. Execution of the conditional statement over boolean expression b, is a demonic choice between two mutators, one assumes b and executes the first branch, another assumes negation of b and executes the second branch. Depending on the value of b and according to the definition of mutator assume, one of these mutators leads to an unreachable outcome. Execution of a while loop iterates the body until the assumption of the guard results in an unreachable outcome. Then it jumps out of the loop body and the negation of the guard is assumed. When calling a routing, the value of the argument is first evaluated and with a store that binds the routine's parameter to the evaluated value, the body starts to be executed. An extra step for calling the routine is also added to the resulting outcome. Execution of sequential commands is sequential composition of execution of each command.

$$
\begin{aligned}
&\mathsf{exec}(x := e) = v \leftarrow \mathsf{eval}(e); x := v \\
&\mathsf{exec}(\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c') = \\
&\quad \mathsf{assume}(b); \mathsf{jump}; \mathsf{exec}(c); \mathsf{jump} \otimes \\
&\quad \mathsf{assume}(\neg b); \mathsf{jump}; \mathsf{exec}(c'); \mathsf{jump} \\
&\mathsf{exec}(\mathbf{while}\ b\ \mathbf{do}\ c) = \\
&\quad (\mathsf{assume}(b); \mathsf{jump}; \mathsf{exec}(c))^*; \mathsf{jump}; \mathsf{assume}_0(\neg b) \\
&\mathsf{exec}(r(e)) = \\
&\quad v \leftarrow \mathsf{eval}(e); \mathsf{call}; \mathsf{with}(\mathbf{0}[x := v], \mathsf{enter}; \mathsf{exec}(c); \mathsf{leave}) \\
&\quad \text{where } \mathbf{routine}\ r(x) = c \\
&\mathsf{exec}(c; c') = \mathsf{exec}(c); \mathsf{exec}(c')
\end{aligned}
$$

Safety of Programs. A command c is considered to be safe with respect to a time bound n, if no execution of that command exceeds n number of cycles, when started from an *empty state* consisting of a store that maps each variable to zero. Notice that the failure outcome is the only outcome that does not satisfy condition $true$. This is formally presented in the following, where $a \triangleright f$ is an alternative to $f(a)$.
$\mathsf{safe\_program}_n\ c = \mathbf{0} \triangleright \mathsf{exec}(c)\ \{\mathrm{true}\}_n$

4 Symbolic Execution

The concrete execution defines the timing behavior of the statements. It, however, cannot serve as an algorithm to verify safety of programs in terms of execution time; for programs that take arbitrary inputs it cannot check all infinit possibilities. To that end, we introduce the *symbolic execution* where *timing specifications* of routines and loops are separately verified and used when verifying the code where a loop or a routine call appears. Along with other functional properties of the program, these specifications can be placed in the loop invariants and routine's contracts. Any violation of these specifications during the symbolic execution causes a verification failure. Accordingly, we enrich the syntax of our original program to support loop invariants (**inv**) and routine contracts, including preconditions (**req**) and postconditions (**ens**) that are of type *assertion*. An assertion can be a boolean expression, availability of e_a amount of time budget denoted by $[e_a]\mathsf{tb}$, a conditional statement over a boolean expression, as well as conjunction of assertions.

Outcome and satisfaction relation are the same as their corresponding ones in the concrete semantic except that the step outcome and time bound in the postcondition are not applicable anymore. The store in the symbolic execution binds each program variable to a *term* that can be a literal number, a *symbol* (ς), representing an arbitrary value, as well as addition, subtraction or multiplication of terms. Accordingly, to constrain the interpretation of symbols in the store, a new component namely *path condition* that is powerset ($\mathcal{P}$) of formulae is added to the state of the execution. A formula is either an equality or an inequality between terms, or the negation of another formula. Note that

the symbolic execution does not intend to provide a time bound, but to verify a provided time bound specified in the precondition of the routine. Hence, we add a third component *heap*, a multi-set of *chunks* that mathematically maps a chunk to a term indicating the number of its repetition in the multi-set, to the state of symbolic execution. We also introduce chunk tb representing one unit of time to maintain the current time budget of the routine in the heap.

$$
\begin{array}{rl}
e_a ::= & z \mid x \mid e_a + e_a \mid e_a - e_a \mid e_a \times e_a \\
b_a ::= & e_a = e_a \mid e_a < e_a \mid \neg b_a \\
a \in \mathit{Assertions} & ::= b_a \mid [e_a]\mathsf{tb} \mid a * a \mid \mathbf{if}\ b_a\ \mathbf{then}\ a\ \mathbf{else}\ a \\
t, \hat{v} \in \mathit{Terms} & ::= z \mid \varsigma \mid t + t \mid t - t \mid t \times t \\
\hat{s} \in \mathit{SStores} & = \mathit{Vars} \to \mathit{Terms} \\
\varphi \in \mathit{Formulae} & ::= t = t \mid t < t \mid \neg\varphi \\
\Phi \in \mathit{PConds} & = \mathcal{P}(\mathit{Formulae}) \\
\mathit{Chunks} & = \{\mathsf{tb}\} \\
\hat{h} \in \mathit{Heaps} & = \mathit{Chunks} \to \mathit{Terms} \\
\mathit{SStates} & = \mathit{PConds} \times \mathit{SStores} \times \mathit{Heaps}
\end{array}
$$

This new representation of the value of variables requires new auxiliary mutators for symbolic execution of commands. The old version of mutator assume, for example, is not applicable anymore; the evaluation of a boolean expression yields a formula rather than a boolean value. To that end, we employ an *SMT solver* to check whether the formula is consistent with the path condition or not. If yes the formula is added to the current path condition. Otherwise, the execution results in an unreachable outcome. Note that the notation $\Phi \vdash_{\mathrm{S}} \varphi$ denotes the SMT solver succeeds in proving that the set of formulae Φ implies the formula φ. Asserting a formula continues the execution if the path condition implies that formula. Otherwise the execution fails. The mutator tbud returns the current time budget.

$$
\begin{array}{l}
\mathsf{assume}(\varphi) = \lambda(\Phi, \hat{s}, \hat{h}).\ \bigotimes \Phi \nvdash_{\mathrm{S}} \neg\varphi.\ \langle (\Phi \cup \{\varphi\}, \hat{s}, \hat{h}) \rangle \\
\mathsf{assert}(b) = \lambda(\Phi, \hat{s}, \hat{h}).\ \bigoplus \Phi \vdash_{\mathrm{S}} [\![b]\!]_{\hat{s}}.\ \langle (\Phi, \hat{s}, \hat{h}) \rangle \\
\mathsf{assume}_0(b) = \hat{s} \leftarrow \mathsf{sstore}; \mathsf{assume}([\![b]\!]_{\hat{s}}) \\
\mathsf{tbud} = \lambda(\Phi, \hat{s}, \hat{h}).\ \langle (\Phi, \hat{s}, \hat{h}), \hat{h}(\mathsf{tb}) \rangle \\
\mathsf{eval}_0(e) = \lambda(\Phi, \hat{s}, \hat{h}).\ \langle (\Phi, \hat{s}, \hat{h}), [\![e]\!]_{\hat{s}} \rangle
\end{array}
$$

We introduce two other important mutators to *produce* and *consume* assertions as follows:

$$
\begin{array}{l}
\mathsf{produce}(b) = \mathsf{assume}_0(b) \\
\mathsf{produce}([e]\mathsf{tb}) = \hat{v} \leftarrow \mathsf{eval}_0(e); \mathsf{assume}_0(\neg\hat{v}{<}0); \\
\quad \lambda(\Phi, \hat{s}, \hat{h}).\langle (\Phi, \hat{s}, \hat{h}[\mathsf{tb}{:=}\hat{h}(\mathsf{tb}){+}\hat{v}]) \rangle \\
\mathsf{produce}(\mathbf{if}\ b\ \mathbf{then}\ a\ \mathbf{else}\ a') = \\
\quad \mathsf{assume}_0(b); \mathsf{produce}(a) \otimes \mathsf{assume}_0(\neg b); \mathsf{produce}(a') \\
\mathsf{produce}(a * a') = \mathsf{produce}(a); \mathsf{produce}(a') \\
\mathsf{consume}(b) = \mathsf{assert}(b) \\
\mathsf{consume}([e]\mathsf{tb}) = \hat{v} \leftarrow \mathsf{eval}_0(e); \mathsf{assert}(\neg\hat{v}{<}0); t \leftarrow \mathsf{tbud}; \\
\quad \mathsf{assert}(\neg t{<}\hat{v}); \lambda(\Phi, \hat{s}, \hat{h}).\langle (\Phi, \hat{s}, \hat{h}[\mathsf{tb}{:=}\hat{h}(\mathsf{tb}){-}\hat{v}]) \rangle \\
\mathsf{consume}(\mathbf{if}\ b\ \mathbf{then}\ a\ \mathbf{else}\ a') = \\
\quad \mathsf{assume}_0(b); \mathsf{consume}(a) \otimes \mathsf{assume}_0(\neg b); \mathsf{consume}(a') \\
\mathsf{consume}(a * a') = \mathsf{consume}(a); \mathsf{consume}(a')
\end{array}
$$

The mutator jump is defined as $\mathsf{consume}([2]\mathsf{tb})$, causing it to consume two cycles when get symbolically executed. This technique can be applied for other mutators mentioned in Sec. 3 too. The symbolic version of other mutators are the same as the ones provided in Sec. 3 except that the state is symbolic. We also define a new mutator $\mathsf{havoc}(\overline{x})$ to bind fresh symbols to the set of variables $\overline{x}$. fresh yields a fresh symbol that is not yet used. It also records this symbol in the path condition for the subsequent requests. The mutator empty resets the heap and block blocks the execution.

$$
\begin{array}{l}
\mathsf{assume}(b) = \mathsf{consume}([\mathsf{cycle}(b){+}1]\mathsf{tb}); \mathsf{assume}_0(b) \\
\mathsf{havoc}(\overline{x}) = \overline{\hat{v}} \leftarrow \mathsf{fresh}; \overline{x} := \overline{\hat{v}} \\
\mathsf{empty} = \lambda(\Phi, \hat{s}, \hat{h}).\ \langle (\Phi, \hat{s}, \mathbf{0}) \rangle \\
\mathsf{block} = \lambda(\Phi, \hat{s}, \hat{h}).\ \top
\end{array}
$$

Replacing concrete mutators by symbolic ones, symbolic execution of assignment, conditional statement, and sequential composition is the same as the concrete execution. In contrast to the concrete execution, in symbolic execution of a routine call, instead of executing the body of a routine it suffices to consume its precondition and then produce its postcondition. As to the execution of the loops, it is treated in two phases; verifying the loop invariant and then using it instead of iterating the body. Verification is done by havocking the target variables (those whose values may change during the execution) of the loop body, emptying the heap, producing the loop invariant, assuming the guard, executing the body and then consuming the invariant. If consumption of the invariant does not fail, it means that the invariant has specified a correct upper-bound for the execution time of the loop statement and the execution gets immediately blocked. In the next phase, execution consumes the invariant, havocs the target variables of the loop, produces the invariant, and then assumes the negation of the guard. We havoc target variables of the loop in both phases.

$$
\begin{array}{l}
\mathsf{sexec}(r(e)) = \hat{v} \leftarrow \mathsf{eval}(e); \mathsf{call}; \\
\quad \mathsf{with}(\mathbf{0}[x := \hat{v}], \mathsf{consume}(a); \mathsf{produce}(a')) \\
\quad \text{where } \mathbf{routine}\ r(x)\ \mathbf{req}\ a\ \mathbf{ens}\ a' \\
\mathsf{scexec}(\mathbf{while}\ b\ \mathbf{inv}\ a\ \mathbf{do}\ c) = \\
\quad \mathsf{consume}(a); \mathsf{havoc}(\mathsf{targets}(c)); \\
\quad (\mathsf{empty}; \mathsf{produce}(a); \mathsf{jump}; \mathsf{assume}(b); \\
\quad\ \mathsf{scexec}(c); \mathsf{jump}; \mathsf{consume}(a); \mathsf{block} \\
\quad\ \otimes \mathsf{jump}; \mathsf{produce}(a); \mathsf{assume}(\neg b))
\end{array}
$$

Safety of Programs. A program is safely executed in a maximum number of n cycles if 1) the execution of the main command starting from a state whose time budget is n does not fail and 2) all of the routines in the program are valid:

$$
\begin{array}{c}
\mathsf{sym\text{-}safe\_program}_n\ c = \\
(\emptyset, \mathbf{0}, \mathbf{0}[\mathsf{tb} := n])\ \triangleright\ \mathsf{sexec}(c)\ \{\mathrm{true}\} \wedge\ (\forall r.\ \mathsf{valid}(r))
\end{array}
$$

A routine is considered to be valid if execution of its body satisfies its contract as follows:

$$\begin{array}{l}\mathsf{valid}(r) = (\emptyset, \mathbf{0}, \mathbf{0}) \ \triangleright \hat{v} \leftarrow \mathsf{fresh}; \mathsf{with}(\mathbf{0}[x := \hat{v}],\\ \quad \mathsf{produce}(a); \mathsf{enter}; \mathsf{sexec}(c); \mathsf{leave});\\ \quad \mathsf{with}(\mathbf{0}[x := \hat{v}], \mathsf{consume}(a'))\ \{\mathrm{true}\}\\ \quad \text{where } \mathbf{routine}\ r(x)\ \mathbf{req}\ a\ \mathbf{ens}\ a' = c\end{array}$$

5 Soundness Proof

In this section we provide a soundness proof of our approach, i.e. if a program is safe in the symbolic execution then it is also safe in the concrete execution. This property can be formulated and proved as follows:

Theorem 1 (Soundness)**.**
$\mathsf{sym\text{-}safe\_program}_n\ c \Rightarrow \mathsf{safe\_program}_n\ c$

Proof. We first introduce an intermediate execution, namely *semi-concrete execution* in which the state of the execution consists of two components; a store that is similar to the store in concrete execution and a *semi-concrete heap* that is similar to the heap in symbolic execution except that it is a function from chunks to natural numbers (and not terms). The mutators assume and assert are defined similar to those of concrete execution and the rest of the mutators resemble their corresponding ones in symbolic execution except for havoc. Instead of fresh symbols, this mutator binds its input variables to some integer numbers that are demonically chosen, meaning that the rest of the execution should not fail for any value for these variables. Using new mutators the semi-concrete execution (scexec) and safety of programs (sc-safe_program) is also defined just like the symbolic version ones. We define validity of the routines as follows where the input parameter of the routine is bound to an integer value that is demonically chosen:

$$\begin{array}{l}\mathsf{sc\text{-}valid}(r) = (\mathbf{0}, \mathbf{0}) \ \triangleright \bigotimes v.\ \mathsf{with}(\mathbf{0}[x := v],\\ \quad \mathsf{produce}(a); \mathsf{enter}; \mathsf{scexec}(c); \mathsf{leave});\\ \quad \mathsf{with}(\mathbf{0}[x := v], \mathsf{consume}(a'))\ \{\mathrm{true}\}\\ \quad \text{where } \mathbf{routine}\ r(x)\ \mathbf{req}\ a\ \mathbf{ens}\ a' = c\end{array}$$

With the aid of this intermediate execution, the soundness of symbolic execution can be proved by incorporating the auxiliary theorems $\mathsf{sym\text{-}safe\_program}_n\ c \Rightarrow \mathsf{sc\text{-}safe\_program}_n\ c$, that can be proved similar to corresponding one in [14], and $\mathsf{sc\text{-}safe\_program}_n\ c \Rightarrow \mathsf{safe\_program}_n\ c$ that follows directly from the soundness of semi-concrete execution of commands formulated in Lemma 2. □

Lemma 2 (Soundness of Semi-concrete execution of commands)**.**
$(\forall r.\ \mathsf{sc\text{-}valid}(r)) \Rightarrow \forall n, c, h, s.\ h \leqslant n \Rightarrow$
$(s, h) \triangleright \mathsf{scexec}(c); \kappa \Rrightarrow (s, h) \triangleright \kappa; \mathsf{exec}(c)$
where, $\phi \Rrightarrow \phi' \Leftrightarrow \forall Q.\ \phi\ \{Q\} \Rightarrow \phi'\ \{Q\}$
and $\kappa = \lambda(s, h).\ \Uparrow_{-h(\mathsf{tb})}\ \langle s \rangle$

Proof. By induction on n. The base case is trivial since the left hand side of the coverage would not hold. Assuming $\forall c, h, s.\ h \leqslant n \Rightarrow (s, h) \triangleright \mathsf{scexec}(c); \kappa \Rrightarrow (s, h) \triangleright \kappa; \mathsf{exec}(c)$, the goal is $\forall c, h, s.\ h \leqslant n+1 \Rightarrow (s, h) \triangleright \mathsf{scexec}(c); \kappa \Rrightarrow (s, h) \triangleright \kappa; \mathsf{exec}(c)$. By nested induction on c and applying the induction hypothesis and the auxiliary semi-concrete lemmas proved in [14] the goal can be established. □

6 Experimental Results

In this section we report on applying our approach on some programs listed in Table(s) 1 and 2. All the test benches, the source code of our compiler for the MSP430 and the verification algorithm developed for this platform can be found at `https://people.cs.kuleuven.be/~jafar.hamin/prext`. Information in Table 1 includes the specified time budget in the contract of the routine (T_{budget}), the number of machine cycles taken when the routine was executed on MSP430 for three different inputs n=10, n=100, and n=1000 ($T_{10}, T_{100}, T_{1000}$), the result and the execution time of the verification algorithm in ms when running on Ubuntu 15.04 with processor Intel at 3.6GHz and 15GB of RAM (R_v, and T_v). These preliminary results indicate that the algorithm does not verify the contracts underestimating the time bound of the routine (see square2 and square4).

Bench	T_{budget}	T_{10}	T_{100}	T_{1000}	R_v	T_v
square1	28n+34	314	2834	28034	✓	304
square2	28n+33	314	2834	28034	×	364
square3	30n+32	314	2834	28034	✓	284
square4	27n+999	314	2834	28034	×	56
loopsum	28n+34	314	2834	28034	✓	264
loopodds	14n+34	174	1434	14034	✓	312
recsum	41n−17	373	3883	38983	✓	128
recisodd	19n+25	203	1832	18023	✓	124
fibonaci	40n−36	364	3964	39964	✓	228

Table 1. Verification versus execution results

We also manually annotated some VeriFast-annotated programs with ghost commands that consume time budget chunks and verified a number of multi-threading and pointer manipulation programs, some listed in Table 2. In addition to the time budget, the contracts of these routines include user-defined predicates [8] to provide the shape of the inputs in the memory. As an example, consider predicate $Tree(\mathbf{struct}\ tree\ ^*t, \mathbf{int}\ depth, \mathbf{int}\ nodes)$ specifying the properties of tree t including the memory permissions, the depth and the number of nodes of that tree. With the aid of

this predicate, the contracts of routines *binTreeSearch* and *parseTree* can be specified as the following:

$$\begin{array}{l}\textbf{bool}\ binTreeSearch(\textbf{struct}\ tree\ ^{*}t, \textbf{int}\ x)\\ \quad \textbf{req}\ Tree(t, ?d, ?n) * 0 \leqslant d * [64 \times d + 27]\mathsf{tb}\\ \quad \textbf{ens}\ Tree(t, d, n)\\ \textbf{int}\ parseTree(\textbf{struct}\ tree\ ^{*}t)\\ \quad \textbf{req}\ Tree(t, ?d, ?n) * 0 \leqslant n * [81 \times n + 24]\mathsf{tb}\\ \quad \textbf{ens}\ Tree(t, d, n)\end{array}$$

Bench	T_{budget}	R_v	T_v
seqSearch	36n+34	✓	644
bubbleSort	$48n^2$+33n+26	✓	664
addMatrix	28mn+33n+35	✓	664
binTreeSearch	64d+27	✓	672
parseTree	81n+24	✓	684
reverseStack	34n+37	✓	660
multiThreadParseTree	600n+800	✓	768
All above together	-	✓	828

Table 2. Time bound verification in VeriFast

Discussion. Although for most cases the verification algorithm provides the result in a reasonable time, for the routines with cubic complexity the SMT solver is not able to do its job in an appropriate time. From the usability point of view, verification of large programs involves more annotations and is more time consuming. Due to modularity of the approach, however, the complexity of annotating such programs remains constant, since each routine is annotated and verified separately. Verification of programs including library calls is still possible provided that timing specification of the routines in the imported libraries is determinable and accessible through a separate header file. Discovering such critical information can be a challenging issue, though. Extending the approach to cover the state of the art optimizing compilers involves defining an accurate concrete execution schema for the high-level programs that is still an open problem.

7 Conclusion

This paper presented a time bound verification approach based on separation logic, where timing behavior along with other non-functional properties of programs are specified in the contracts of the routines and loops. We provided a soundness proof of the approach for the microcontroller MSP430 and our non-optimizing compiler, and tested it on a number of multi-threading and pointer manipulation programs.

8 Acknowledgements

This research was funded by the Flemish Research Fund (grant G.0058.13).

References

[1] E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. Verified resource guarantees using costa and key. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 73–76. ACM, 2011.

[2] R. M. Amadio, N. Ayache, F. Bobot, J. P. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, et al. Certified complexity (cerco). In *Foundational and Practical Aspects of Resource Analysis*, pages 1–18. Springer, 2013.

[3] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

[4] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr. The auspicious couple: Symbolic execution and wcet analysis. In *OASIcs-OpenAccess Series in Informatics*, volume 30. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[5] M. De Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 161–166. IEEE, 2008.

[6] J. Hoffmann and Z. Shao. Type-based amortized resource analysis with integers and arrays. *Journal of Functional Programming*, 25:e17, 2015.

[7] T. Instruments. Msp430x5xx/msp430x6xx family users-guide. *URL: http://www. ti. com/lit/ug/slau208j/slau208j. pdf*, 2012.

[8] B. Jacobs, J. Smans, and F. Piessens. The verifast program verifier: A tutorial, 2014.

[9] Y.-K. Kim, W. Shin, and C.-H. Chang. Design of static execution time analyzer using partial path. In *Systems and Informatics (ICSAI), 2012 International Conference on*, pages 2480–2483. IEEE, 2012.

[10] J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic loop bound computation for wcet analysis. In *Perspectives of Systems Informatics*, pages 227–242. Springer, 2011.

[11] K. Nakata and T. Uustalu. A hoare logic for the coinductive trace-based big-step semantics of while. pages 488–506, 2010.

[12] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[13] P. Tranquilli. Indexed labels for loop iteration dependent costs. *arXiv preprint arXiv:1306.2692*, 2013.

[14] F. Vogels, B. Jacobs, , and F. Piessens. Featherweight verifast. *Logical Methods in Computer Science*, 11(3):1–57, 2015.